**Task1(a):**

From 'input1a.txt' we get nodes and edges and constructs an adjacency matrix for the graph, and writes the matrix to 'output1a.txt'. The input file format consists of the number of nodes and edges on the first line, followed by lines representing edges with their source, destination, and weight. The code creates a square matrix to represent the graph, where each cell (i, j) contains the weight of the edge from node i to node j. The resulting adjacency matrix is written to the output file.

**Task1(b):**

From 'input1b.txt' we get nodes and edges and constructs an adjacency list representation for the graph, and writes the adjacency list to 'output1b.txt'. The input file format consists of the number of nodes and edges on the first line, followed by lines representing edges with their source, destination, and weight. The code creates an adjacency list where each node is represented by a list containing tuples for its neighbors and edge weights. The resulting adjacency list is then written to the output file.

**Task 2:**

The number of nodes and edges from the first line of 'input2.txt' and initializes an empty graph as a list of lists. It reads the remaining lines, each containing two integers (u and v), representing an undirected edge between nodes u and v, and adds them to the 'graph' as bidirectional edges. The BFS starts from node 1 and explores the graph, maintaining a queue 'q' to store the nodes to be visited. It uses the 'explore' list to keep track of visited nodes. The order of visited nodes is written to 'output2.txt' as the BFS progresses. In summary, this code performs BFS on an undirected graph starting from node 1 and records the order of visited nodes in 'output2.txt'.

**Task 3:**

The code performs a depth-first search (DFS) starting from node 1 in an undirected graph represented using an adjacency list. It reads the graph information from 'input3.txt', explores the graph using DFS, and writes the order of visited nodes to 'output3.txt'. It reads the remaining lines, each containing two integers (u and v), representing an undirected edge between nodes u and v, and adds them to the 'graph' as bidirectional edges. The DFS starts from node 1, initializes a 'explore' list to keep track of visited nodes, and uses a recursive function 'DFS' to explore the graph. The function marks the current node as visited, writes its value to 'output3.txt', and then recursively explores its unvisited neighbors.

**Task 4:**

The code checks for the presence of cycles in a directed graph represented using an adjacency list. It reads the graph information from 'input4.txt', performs a cycle detection algorithm based on Depth-First Search (DFS), and writes either 'YES' or 'NO' to 'output4.txt' depending on the presence of cycles. The DFS-based function 'DFS' checks for cycles in the graph starting from a given node 'u'. It maintains two boolean lists, 'explored' and 're\_explored,' to track visited nodes and nodes in the current DFS path. If it encounters a node that is already being explored (re\_explored), it identifies a cycle in the graph and returns True. The 'cycle' function iterates through all nodes of the graph and runs DFS on unexplored nodes to detect cycles. If any cycle is found, it returns True, indicating that the graph contains cycles; otherwise, it returns False.The main part of the code calls the 'cycle' function with the graph and number of nodes as arguments. If a cycle is detected, it writes 'YES' to 'output4.txt', otherwise 'NO'.

**Task 5:**

The code finds the shortest path from node 's' to node 'd' in an undirected graph using Breadth-First Search (BFS). It reads the graph information from 'input5.txt', calculates the minimum time (number of edges) to reach 'd' from 's', and writes the minimum time and the shortest path to 'output5.txt'. The code reads 'n', 'm', and 'd' from the first line of 'input5.txt', where 'n' represents the number of nodes, 'm' is the number of edges, and 'd' is the destination node. It also initializes 's' to 1, indicating the starting node of the search. It creates an empty graph as a list of lists to represent the adjacency list, and then reads the edges from the subsequent lines of 'input5.txt' and populates the 'graph'. The BFS starts from the starting node 's', using a queue 'q' to explore the graph level by level. It calculates the shortest time to reach each node and constructs the shortest path to each node in the 'path' list. After BFS is complete, the code writes the minimum time to reach node 'd' in the first line and the shortest path to 'd' in the second line of 'output5.txt'.

**Task 6:**

The code reads a grid-shaped graph from 'input6.txt', where each cell can be either empty, denoted by ".", or contain a diamond, denoted by "D". The code then finds the maximum number of diamonds that can be collected in a single connected region of the grid. The 'adj' function defines the neighboring cells of a given cell (i, j) that are within the grid and contain an empty space (not "#"). It returns a list of valid neighbor tuples. The DFS-based function 'dfs' explores the connected region starting from a cell (i, j). It updates the 'matrix' to mark visited cells and counts the number of diamonds (marked as "D") encountered during the exploration. The code then iterates through all cells in the grid, calling 'dfs' for each non-blocked cell (not "#") and keeps track of the maximum number of diamonds collected in a single connected region. The final result is written to 'output6.txt'.